Journal of Logical and Algebraic Methods in Programming 84 (2015) 197–217

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

A framework for computing finite SLD trees [☆]

Naoki Nishida [a], Germán Vidal [b,*]

[a] Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
[b] MiST, DSIC, Universitat Politècnica de València, Camino de Vera, s/n, 46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 7 May 2014
Received in revised form 30 November 2014
Accepted 30 November 2014
Available online 5 December 2014

Keywords:
Logic programming
Semantics
Program analysis

ABSTRACT

The search space of SLD resolution, usually represented by means of a so-called SLD tree, is often infinite. However, there are many applications that must deal with possibly infinite SLD trees, like partial evaluation or some static analyses. In this context, being able to construct a finite representation of an infinite SLD tree becomes useful.

In this work, we introduce a framework to construct a finite data structure representing the (possibly infinite) SLD derivations for a goal. This data structure, called *closed* SLD tree, is built using four basic operations: unfolding, flattening, splitting, and subsumption. We prove some basic properties for closed SLD trees, namely that both computed answers and calls are preserved. We present a couple of simple strategies for constructing closed SLD trees with different levels of abstraction, together with some examples of its application. Finally, we illustrate the viability of our approach by introducing a test case generator based on exploring closed SLD trees.

1. Introduction

In the context of logic programming, *partial evaluation* (also known as partial deduction [25]) is a well known technique to specialize programs. Intuitively speaking, given a logic program P and a finite set of atoms $\mathcal{A} = \{a_1, \ldots, a_n\}$, one should construct finite—possibly *incomplete*—SLD trees for the atomic goals $\leftarrow a_1, \ldots, \leftarrow a_n$, such that every leaf in these trees is either successful, a failure, or only contains atoms that are instances of $\{a_1, \ldots, a_n\}$. This condition, called *closedness* condition [25], ensures that the computed trees are *self-contained*, which in turn guarantees the correctness of the approach.

In this work, we aim at generalizing this idea by introducing a general framework to construct finite representations of (possibly infinite) SLD trees, so that they can also be used in other contexts. Indeed, there are many problems in computer science that require dealing with (a finite representation of) an infinite search space. Besides partial evaluation, static analyses and model checking techniques, for instance, aim at exploring all possible computations starting from a goal or from a class of goals, some of which are usually infinite. Furthermore, there are many approaches (see, e.g., [1–3,9,8,15,16,27,29,38]) that advocate a *transformational approach* in which a program with imperative, object-oriented or concurrent features—which are often difficult to analyze—are compiled into a simpler, rule-based intermediate language (usually ignoring some details or abstracting away some features), where rigorous program analyses can be defined and implemented in a simpler

☆ This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* (Secretaría de Estado de Investigación, Desarrollo e Innovación) under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.

* Corresponding author.
E-mail addresses: nishida@is.nagoya-u.ac.jp (N. Nishida), gvidal@dsic.upv.es (G. Vidal).

http://dx.doi.org/10.1016/j.jlamp.2014.11.006
2352-2208/© 2014 Elsevier Inc. All rights reserved.

way. One of the most popular such rule-based languages is Prolog. Therefore, our framework may contribute to the use of Prolog as a target language within the above transformational approach to program analysis.

In particular, we introduce an extension of standard SLD trees that are built using four basic operations: unfolding (based on SLD resolution), flattening (i.e., generalizing some atoms in a goal), splitting (i.e., partitioning a goal into a number of subgoals that are then evaluated independently) and subsumption (a sort of memoization). When all the leaves of the tree are either successful, a failure or a subsumed goal, we speak of a *closed* SLD tree. Besides formalizing this notion, our main contributions are the following:

- Given a program and a goal, we show that a closed SLD tree can always be constructed using an appropriate strategy. Here, we present a couple of strategies that produce closed SLD trees with different levels of abstraction.
- We prove that the computed answers of a standard (possibly infinite) SLD tree and those of an associated closed SLD tree (no matter the considered strategy) are the same. Thus, the abstraction involved in producing closed SLD trees does not affect the computed answers represented by the tree.
- We also prove that, for every call in a standard SLD tree, there is a (possibly more general) call in any associated closed SLD tree. This property guarantees the usefulness of closed SLD trees for those program analysis where computing (an approximation of) the call patterns of a goal is required.
- Then, we show how the success set of a closed SLD tree can be represented in a compact way by means of equations. This might be useful, e.g., for program comprehension.
- Finally, we illustrate the viability of our approach by introducing a fully automatic test case generator based on exploring closed SLD trees.

This paper is organized as follows. After some preliminaries, Section 3 introduces the basic operations involved in the construction of closed SLD trees and states a number of properties for them. Specific strategies for constructing closed SLD trees are introduced and illustrated by means of examples in Section 4. In Section 5, the design of a test case generator is introduced. Finally, Section 6 discusses some related work and Section 7 concludes and points out some directions for further research.

2. Preliminaries

In this section, we briefly present some basic notions from logic programming; we refer the interested reader to [24] for a detailed introduction to this paradigm.

We consider a first-order language with a fixed vocabulary of predicate symbols, function symbols, and variables denoted by Π, $\mathcal{F}$, and $\mathcal{V}$, respectively. We let $\mathcal{T}(\mathcal{F},\mathcal{V})$ denote the set of *terms* constructed using symbols from $\mathcal{F}$ and variables from $\mathcal{V}$. An *atom* has the form $p(t_1,\ldots,t_n)$ with $p/n \in \Pi$ and $t_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$ for $i=1,\ldots,n$. A *definite clause* has the form *head* $\leftarrow$ *body*, where *head* is an atom and *body* $\equiv (a_1 \wedge \cdots \wedge a_n)$ is a goal[1] (a conjunction of atoms); the empty goal is denoted by *true*. We usually denote goals with capital letters (e.g., $G, C, \ldots$) and atoms with small letters (e.g., $a, b, \ldots$). A *definite program* is a finite set of definite clauses. In the following, we focus on definite programs and will refer to them just as programs (analogously to clauses and goals). $\mathcal{V}ar(s)$ denotes the set of variables in the syntactic object s.

Substitutions and their operations are defined as usually. In particular, a substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ denotes a (partial) mapping σ such that $\sigma(x) = t_i$ if $x = x_i$, $i = 1,\ldots,n$, and $\sigma(x) = x$ otherwise. The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is called the *domain* of a substitution σ. The empty substitution is denoted by *id*. A substitution σ is idempotent if $\sigma = \sigma \cdot \sigma$, where "$\cdot$" denotes the standard composition operator on substitutions. The *restriction* $\theta \upharpoonright_V$ of a substitution θ to a set of variables V is defined as follows: $x\theta \upharpoonright_V = x\theta$ if $x \in V$, and $x\theta \upharpoonright_V = x$ otherwise. We say that $\theta = \sigma\ [V]$ if $\theta \upharpoonright_V = \sigma \upharpoonright_V$. This notation is extended to sets of substitutions in the natural way: $\Theta_1 = \Theta_2\ [V]$ implies that there is a substitution $\theta_1 \in \Theta_1$ iff there is a substitution $\theta_2 \in \Theta_2$ such that $\theta_1 = \theta_2\ [V]$. A syntactic object s_1 is *more general* than a syntactic object s_2, denoted $s_1 \leq s_2$, if there exists a substitution θ such that $s_2 = s_1\theta$. A substitution θ is a *unifier* of two syntactic objects t_1 and t_2 iff $t_1\theta = t_2\theta$; furthermore, θ is the *most general unifier* of t_1 and t_2, denoted by $\mathsf{mgu}(t_1 = t_2)$ if, for every other unifier σ of t_1 and t_2, we have that $\theta \leq \sigma$. The mgu operator is naturally extended to a conjunction of equations. A *variable renaming* is a substitution that is a bijection on $\mathcal{V}$. Two syntactic objects t_1 and t_2 are *variants* (or equal up to variable renaming), denoted $t_1 \approx t_2$, if $t_1 = t_2\rho$ for some variable renaming ρ.

Computations in logic programming are formalized by means of SLD resolution. The notion of *computation rule* $\mathcal{R}$ is used to select an atom within a goal for its evaluation. Given a program P, a goal $G \equiv (a_1 \wedge \cdots \wedge a_n)$, and a computation rule $\mathcal{R}$, we say that $G \leadsto_{P,\mathcal{R},\sigma} G'$ is an *SLD resolution step* for G with P and $\mathcal{R}$ if $\mathcal{R}(G) = a_i$, $1 \leq i \leq n$, is the selected atom, $h \leftarrow b_1 \wedge \cdots \wedge b_m$ is a renamed apart clause (i.e., a clause with fresh variables not used before in the computation) of P, $\sigma = \mathsf{mgu}(a_i = h)$, and $G' \equiv (a_1 \wedge \cdots \wedge a_{i-1} \wedge b_1 \wedge \cdots \wedge b_m \wedge a_{i+1} \wedge \cdots \wedge a_n)\sigma$; we often omit P, $\mathcal{R}$, and/or σ in the notation of an SLD resolution step when they are clear from the context.

Let P be a program, G_0 a goal and $\mathcal{R}$ a computation rule. An *SLD derivation* for G_0 with P and $\mathcal{R}$ is a (finite or infinite) sequence $G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of renamed apart clauses of P, a sequence $\mathcal{R}(G_0), \mathcal{R}(G_1), \mathcal{R}(G_2), \ldots$

[1] We use "$\equiv$" to denote the identity on syntactic objects, and identify goals with (possibly empty) conjunctions of atoms.